ACH 2147 — DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

ARQUITETURAS E PROCESSOS DE SISTEMAS DISTRIBUÍDOS

Daniel Cordeiro

4 e 6 de abril de 2017

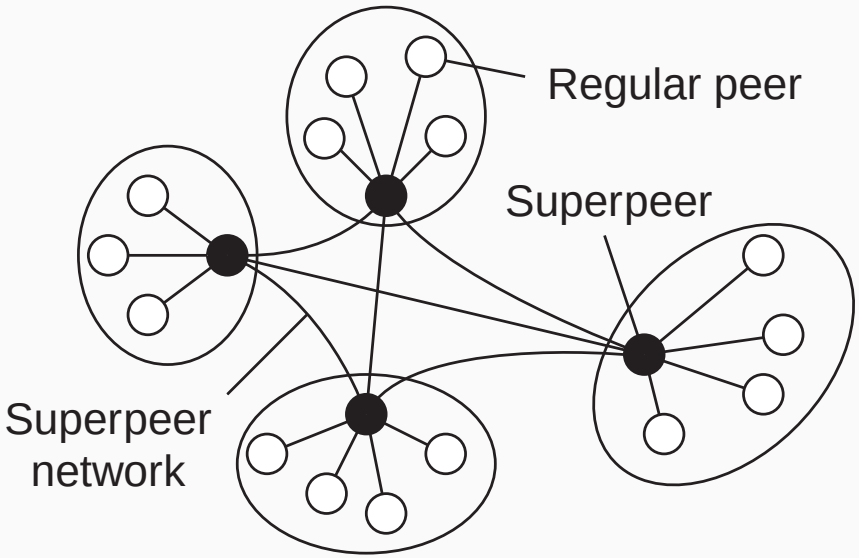
Escola de Artes, Ciências e Humanidades | EACH | USP

Enunciado do EP está no ar.

Prazo para entrega: 22 de abril às 23h55

Observação

Às vezes, selecionar alguns nós para realizar algum trabalho específico pode ser útil.



Exemplos:

- Peers para manter um índice (para buscas)
- Peers para monitorar o estado da rede
- Peers capazes de configurar conexões

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

A está atrás de um firewall, B está na Internet pública

- A configura uma conexão TCP (para os pacotes de controle) com um superpeer S
- S configura uma conexão TCP (para redirecionar os pacotes de controle) com B
- A chamada real usa pacotes UDP diretamente entre A e B

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

A está atrás de um firewall, B está na Internet pública

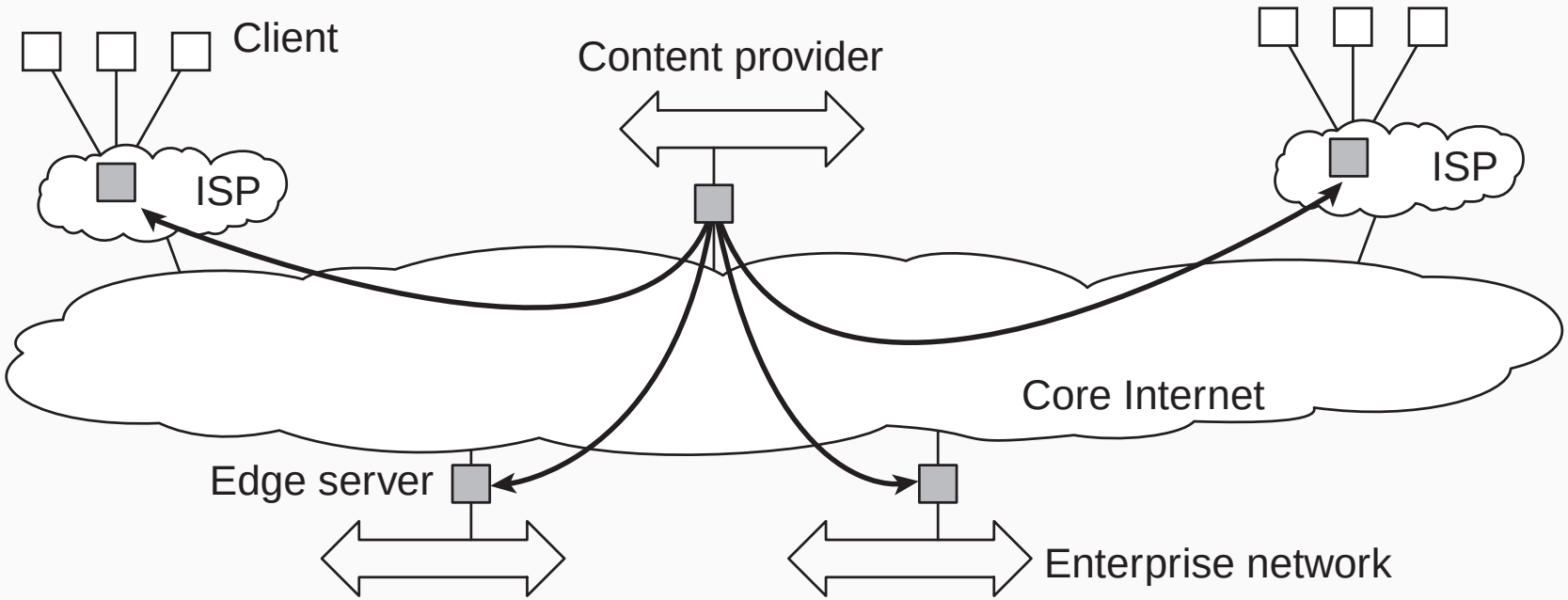
- A configura uma conexão TCP (para os pacotes de controle) com um superpeer S
- S configura uma conexão TCP (para redirecionar os pacotes de controle) com B
- A chamada real usa pacotes UDP diretamente entre A e B

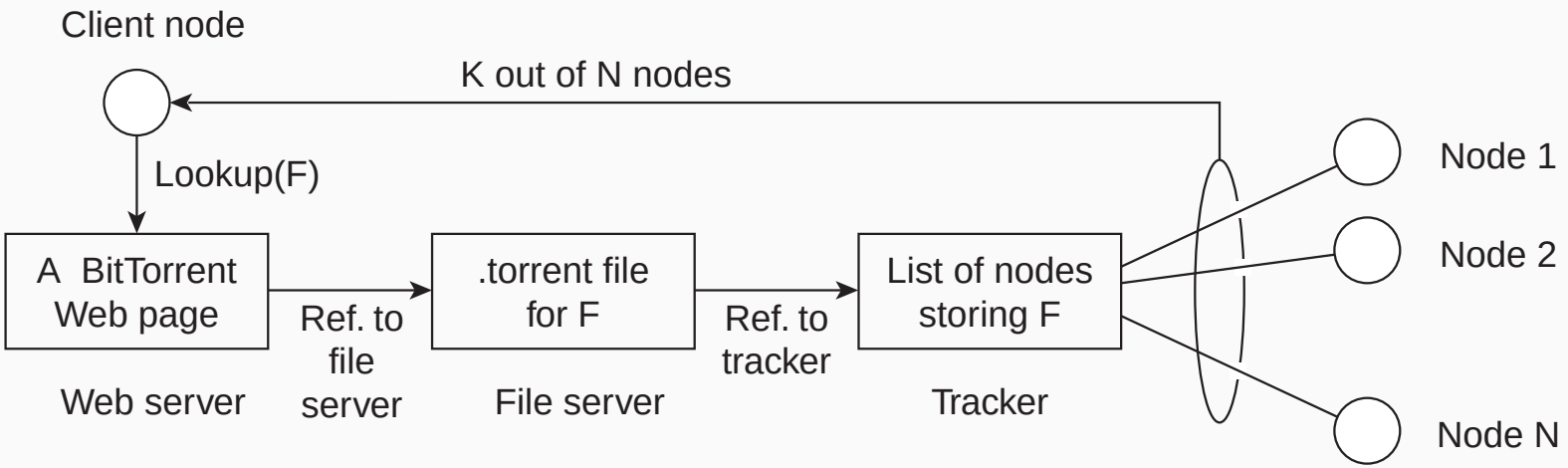
Tanto A quanto B estão atrás de um firewall

- A conecta com um superpeer S via TCP
- S configura uma conexão TCP com B
- Para a chamada real, outro superpeer é usado para funcionar como retransmissor (relay): A (e B) configura a conexão com R
- A chamada é encaminhada usando duas conexões TCP, usando R como intermediário

Exemplo:

Arquiteturas de servidores de borda (*edge-server*), utilizados com frequência como Content Delivery Networks (redes de distribuição de conteúdo).





Ideia básica

Assim que um nó identifica de onde o arquivo será baixado, ele se junta a uma swarm (multidão) de pessoas que, em paralelo, receberão pedaços do arquivo da fonte e redistribuirão esses pedaços entre os outros.

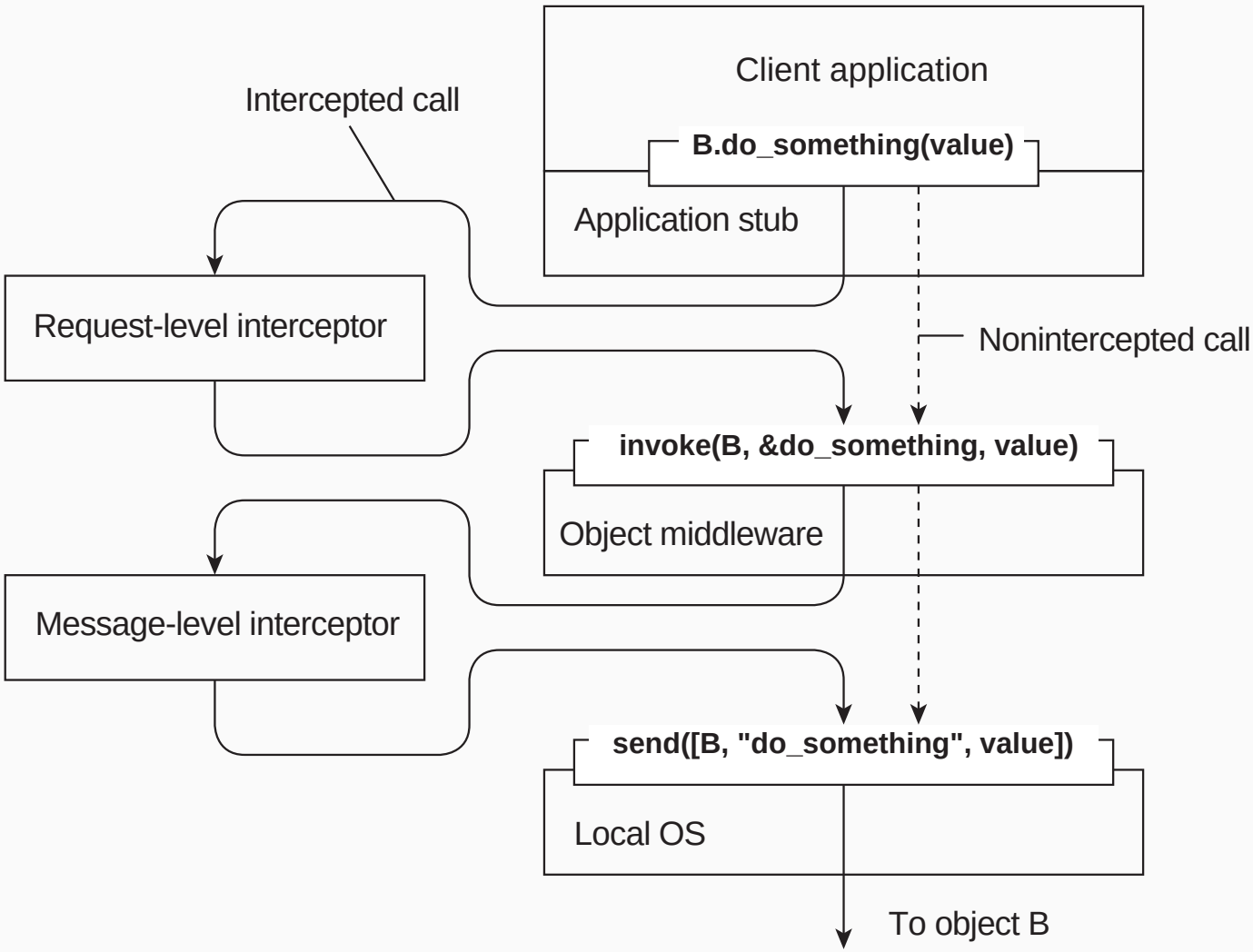
Problema

Em muitos casos, arquiteturas/sistemas distribuídos são desenvolvidos de acordo com um estilo arquitetural específico. O estilo escolhido pode não ser o melhor em todos os casos $\Rightarrow$ é necessário adaptar o comportamento do middleware (dinamicamente).

Interceptors

Interceptam o fluxo de controle normal quando um objeto remoto for invocado.

INTERCEPTORS



- Separação de interesses: tente separar as funcionalidades extras e depois costurá-las em uma única implementação ⇒ aplicabilidade restrita (*toy examples*)
- Reflexão computacional: deixe o programa inspecionar-se em tempo de execução e adaptar/mudar suas configurações dinamicamente, se necessário ⇒ ocorre principalmente no nível da linguagem, aplicabilidade não é muito clara.
- Projeto baseado em componentes: organize uma aplicação distribuída em componentes que podem ser substituídos dinamicamente quando necessário ⇒ causa muitas e complexas interdependências entre componentes.

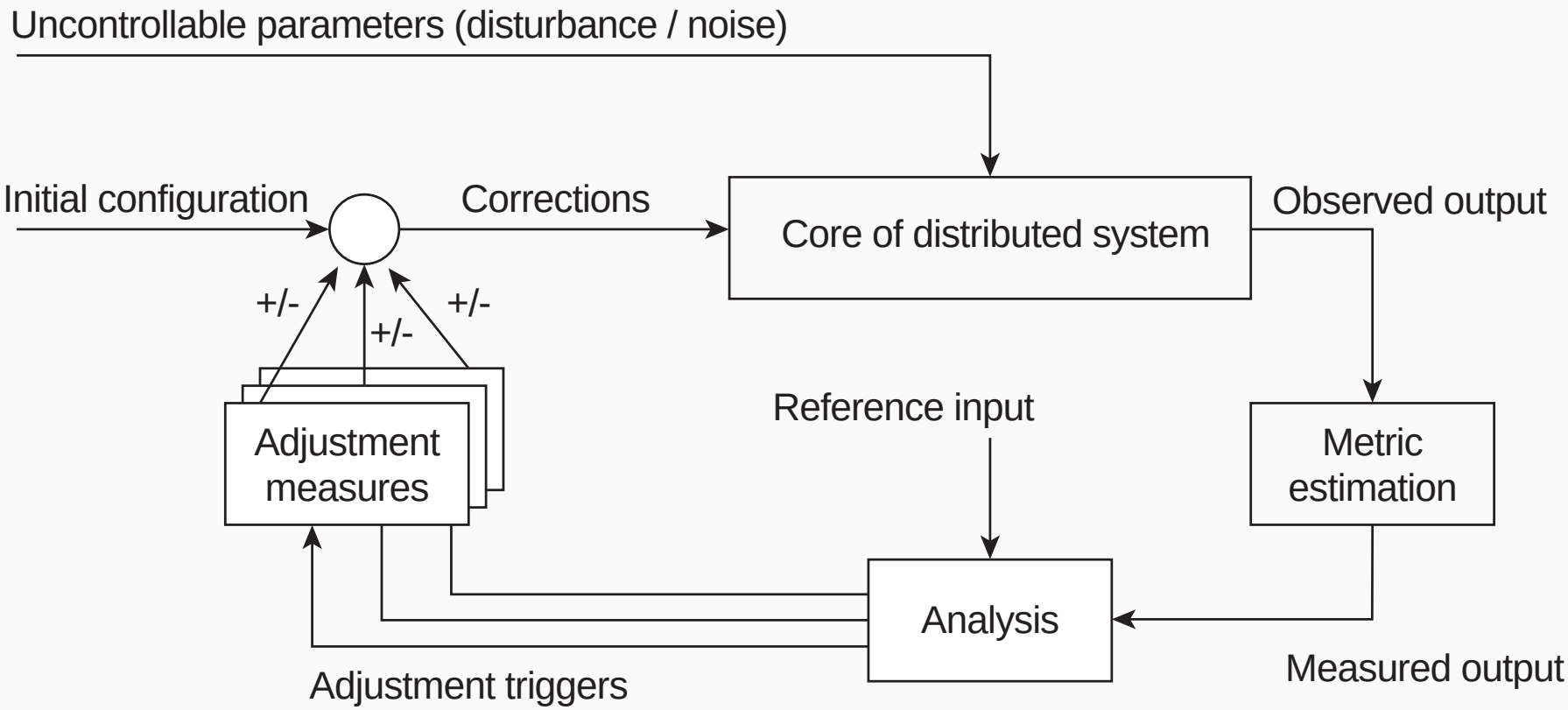
Observação

A distinção entre arquiteturas de sistemas e arquiteturas de software fica confusa quando adaptação automática deve ser considerada:

- Autoconfiguração
- Autogerenciamento
- Autocura
- Auto-otimização
- Auto-*

MODELO DE REGULAÇÃO POR FEEDBACK

Em muitos casos, sistemas auto-* são organizados como um sistema de regulação por feedback



Processos

Ideia básica

Construir um processador virtual com software, em cima dos processadores físicos:

processador: (hardware) provê um conjunto de instruções junto com a capacidade de executá-las automaticamente

thread: (software) um processador mínimo com um contexto que possui uma série de instruções que podem ser executados. Gravar o contexto de uma thread implica em parar a execução e guardar todos os dados necessários para continuar a execução posteriormente

processo: (software) um processador em cujo contexto pode ser executado uma ou mais threads. Executar uma thread significa executar uma série de instruções no contexto daquela thread.

Contextos

Contexto do processador: um conjunto mínimo de valores guardados nos registradores do processador, usado para a execução de uma série de instruções (ex: ponteiro de pilha, registradores, contador de programa, etc.)

Contexto de thread: um conjunto mínimo de valores guardado em registradores e memória, usado para a execução de uma série de instruções (i.e., contexto do processador e estado)

Contexto de processo: um conjunto mínimo de valores guardados em registradores e memória, usados para a execução de uma thread (i.e., contexto de threads e os valores dos registradores de MMU — Memory Management Unit)

TROCA DE CONTEXTO

Observações:

1. Threads compartilham o mesmo espaço de endereçamento. A realização da troca de contexto pode ser feita independentemente do sistema operacional
2. A troca de processos é mais custosa, já que envolve o sistema operacional
3. Criar e destruir threads é muito mais barato do que fazer isso com processos

Problema:

O núcleo do sistema operacional deve prover threads, ou elas devem ser implementadas em nível de usuário?

Solução no nível de usuário

- Todas as operações podem ser realizadas dentro de um único processo — **muito** mais eficiente
- Todos os serviços providos pelo núcleo são feitos *em nome do processo na qual a thread reside* — se o núcleo decidir bloquear a thread, o processo inteiro será bloqueado
- Threads são usadas quando há muitos eventos externos: *threads são bloqueadas com base nos eventos recebidos* — se o kernel não puder distinguir as threads, como permitir a emissão de sinais do SO para elas?

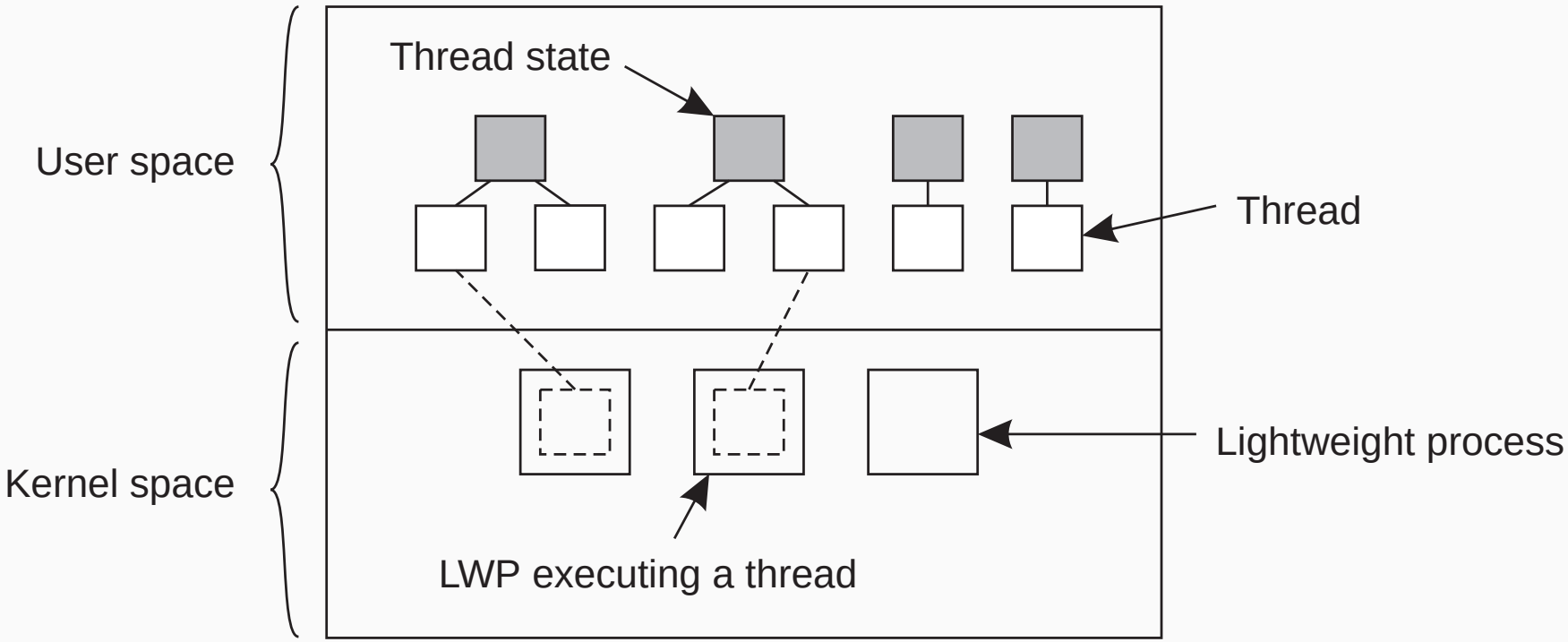
Solução no nível do SO

- O núcleo contém a implementação do software de threading. *Todas* as operações são chamadas de sistemas
- Operações que bloqueiam uma thread não são mais um problema: o núcleo escalona outra thread ociosa dentro do mesmo processo
- Tratamento de eventos externos mais simples: o núcleo (que recebe todos os eventos) escalona a thread associada com aquele evento
- O problema é a perda de eficiência devido ao fato de que todas as operações em threads requerem um *trap* pro núcleo

Conclusão

O melhor é tentar juntar threads de nível de usuário e de nível do SO em um único conceito.

Introduz uma abordagem em dois níveis para threads: processos leves que podem ser executar threads de nível de usuário



Operação principal

- uma thread de nível de usuário realizam uma chamadas de sistema: o LWP (*light-weight process)* que estiver executando aquela thread bloqueia. A thread continua associada àquele LWP.
- O núcleo pode escalonar outro LWP com uma thread associada pronta para execução. Essa thread pode ser trocada por qualquer outra thread de nível de usuário que esteja pronta.
- Uma thread executa uma operação de nível de usuário bloqueante — faça troca de contexto para uma thread pronta (e então a associe ao mesmo LWP).
- Quando não há threads para executar, um LWP pode ficar ocioso, e mesmo destruído pelo núcleo.

Clientes web multithreaded — escondem a latência da rede:

- Navegador analisa a página HTML sendo recebida e descobre que *muitos outros arquivos devem ser baixados*.
- Cada arquivo é baixado por uma thread separada; cada uma realiza uma requisição HTTP (bloqueante)
- A medida que os arquivos chegam, o navegador os exibem.

Múltiplas chamadas requisição–resposta (RPC) para outras máquinas

- Um cliente faz várias chamadas simultâneas, cada uma em uma thread diferente
- Ele espera até que todos os resultados tenham chegado.
- Obs: se as chamadas são a servidores diferentes, você pode ter um speed-up linear

Melhorias no desempenho

- Iniciar uma thread é **muito** mais barato do que iniciar um novo processo
- Ter servidores single-threaded impedem o uso de sistemas multiprocessados
- Tal como os clientes: esconda a latência da rede reagindo à próxima requisição enquanto a anterior está enviando sua resposta.

Melhorias na estrutura

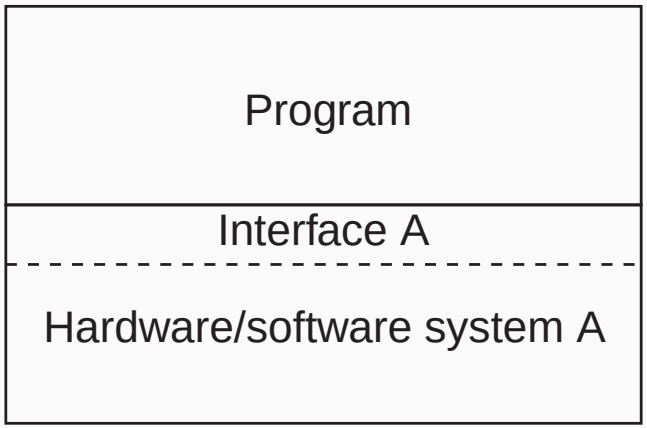
- A maioria dos servidores faz muita E/S. Usar chamadas bloqueantes simples e bem conhecidas simplifica o programa.
- Programas multithreaded tendem a ser menores e mais fáceis de entender, já que simplificam o fluxo de controle.

Virtualização

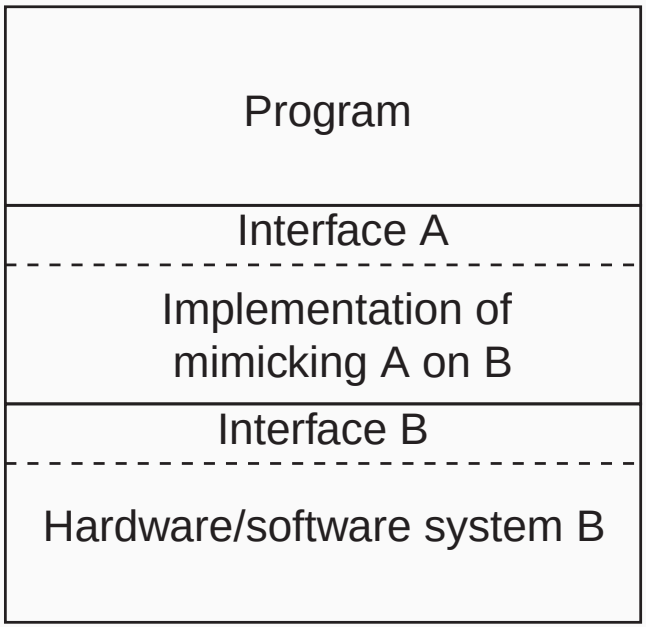
VIRTUALIZAÇÃO

É cada vez mais importante:

- Hardware muda mais rápido do que software
- Melhora a portabilidade e a migração de código
- Provê isolamento de componentes com falhas ou sendo atacados

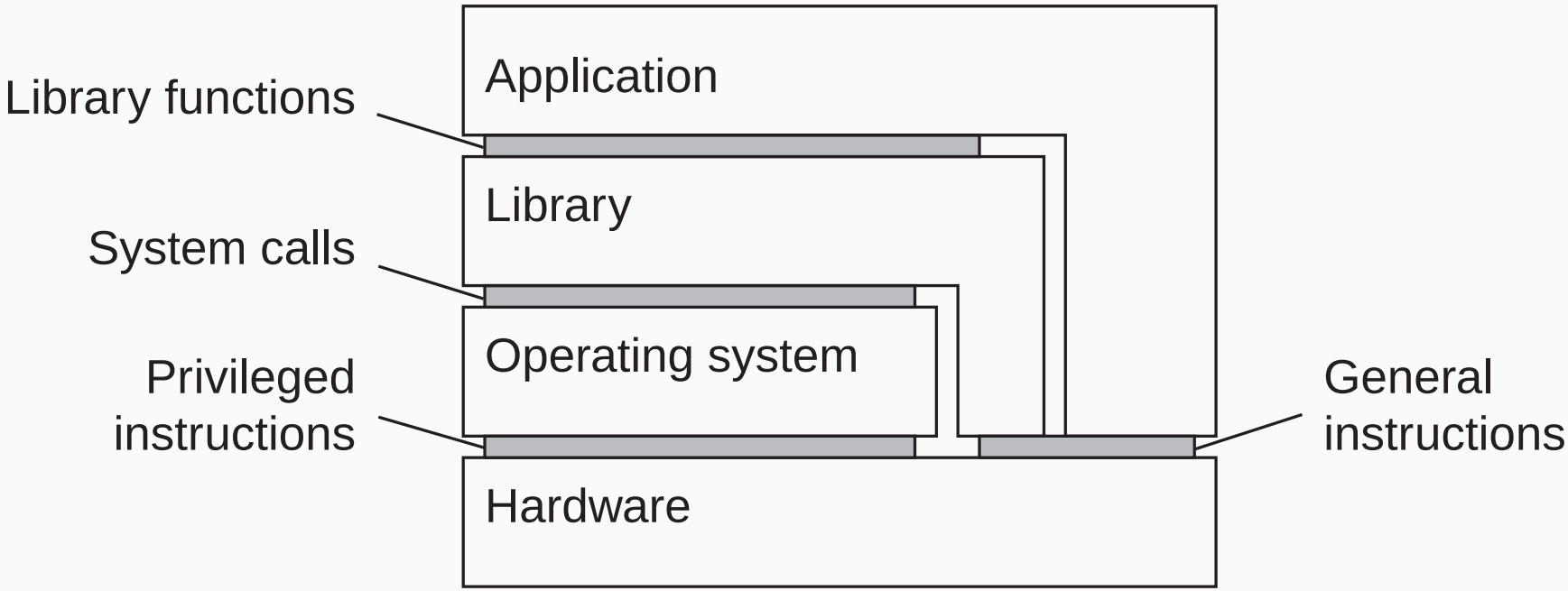


(a)

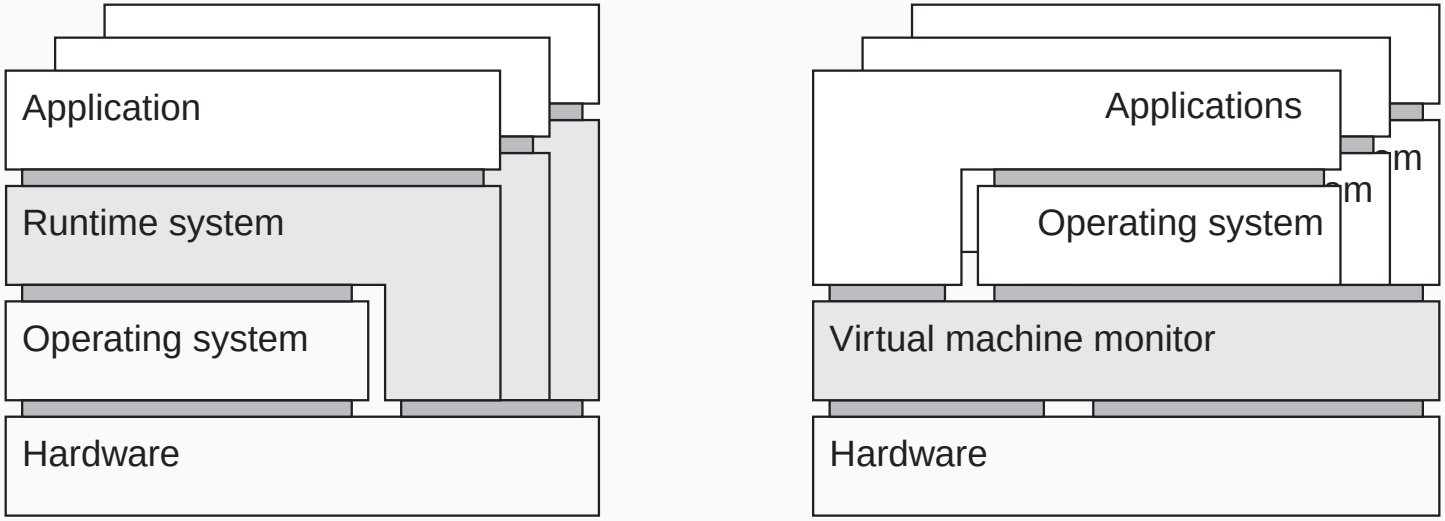


(b)

Virtualização pode ocorrer em diferentes níveis, dependendo das interfaces oferecidas pelos diferentes componentes do sistema:



PROCESSOS VMS VS. MONITORES DE VM



- **Processos VMs**: um programa é compilado para um código intermediário (portátil) que é executado por um interpretador. Ex: Java VM.
- **Monitor de VM**: uma camada de software que imita o conjunto de instruções do hardware — pode executar um sistema operacional completo e suas aplicações. Ex: VMware, VirtualBox.

MONITORES DE VM EM SISTEMAS OPERACIONAIS

Monitores de VM são executadas em cima de sistemas operacionais existentes.

- Realizam tradução binária: enquanto executam uma aplicação ou sistema operacional, traduzem as instruções para as instruções da máquina física
- Distinguem instruções sensíveis: traps para o núcleo original (system calls ou instruções privilegiadas)
- Instruções sensíveis são substituídas por chamadas ao Monitor de VM